LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation

Chris Lattner Vikram Adve
University of Illinois at Urbana-Champaign
{lattner,vadve}@cs.uiuc.edu
http://llvm.cs.uiuc.edu/

ABSTRACT

This paper describes LLVM (Low Level Virtual Machine), a compiler framework designed to support *transparent, lifelong program analysis and transformation* for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and offline. LLVM defines a common, low-level code representation in Static Single Assignment (SSA) form, with several novel features: a simple, *language-independent* type-system that exposes the primitives commonly used to implement high-level language features; an instruction for typed address arithmetic; and a simple mechanism that can be used to implement the exception handling features of high-level languages (and `setjmp`/`longjmp` in C) uniformly and efficiently. The LLVM compiler framework and code representation together provide a combination of key capabilities that are important for practical, lifelong analysis and transformation of programs. To our knowledge, no existing compilation approach provides all these capabilities. We describe the design of the LLVM representation and compiler framework, and evaluate the design in three ways: (a) the size and effectiveness of the representation, including the type information it provides; (b) compiler performance for several interprocedural problems; and (c) illustrative examples of the benefits LLVM provides for several challenging compiler problems.

1. INTRODUCTION

Modern applications are increasing in size, becoming more dynamic, change behavior throughout their execution, and often have components written in multiple different languages. While some applications have small hot spots, others spread their execution time evenly throughout the application [11]. In order to provide maximum efficiency for all of these programs, we believe that program analysis and transformation must be performed throughout the lifetime of the program. Such "lifelong code optimization" techniques encompass interprocedural optimizations performed at link-time (to preserve the benefits of separate compilation), machine-dependent optimizations at install time on each system, dynamic optimization at runtime, and profile-guided optimization between runs ("idle-time") using profile information collected from the end-user.

Program optimization is not the only use for lifelong analysis and transformation. Other emerging applications of static analysis are fundamentally interprocedural, and are therefore most convenient to perform at link-time (examples include static debugging, static leak detection [22], and memory management transformations [26]). Sophisticated analyses and transformations are being developed to enforce program safety, but must be done at software installation time or load-time [17]. Allowing lifelong reoptimization of the program gives architects the power to evolve processors and exposed interfaces in more flexible ways [10, 18], while allowing legacy applications to run *well* on new systems.

This paper presents **LLVM** — Low-Level Virtual Machine — a compiler framework that aims to make lifelong program analysis and transformation available for arbitrary software, and in a manner that is transparent to programmers. LLVM achieves this through two parts: (a) *a code representation* with several novel features that serves as a common representation for analysis, transformation, and code distribution; and (b) *a compiler design* that exploits this representation to provide a combination of capabilities that is not available in any previous compilation approach we know of.

The LLVM code representation describes a program using an abstract RISC-like instruction set but with key higher-level information for effective analysis, including type information, explicit control flow graphs, and an explicit dataflow representation (using an infinite, typed register set in Static Single Assignment form [14]). There are several novel features in the LLVM code representation: (1) A low-level, *language-independent type system* that can be used to *implement* data types and operations from high-level languages, exposing their implementation behavior to all stages of optimization. (2) Instructions for performing type conversions and low-level address arithmetic while preserving type information. (3) Two low-level exception-handling instructions for implementing language-specific exception semantics, while exposing exceptional control flow to the compiler explicitly. (4) A simple, low-level memory model distinguishing heap, stack, global data, and code memory regions, accessed through typed pointers.

The LLVM representation is *source-language-independent*, for two reasons. First, it uses a low-level instruction set and memory model, only slightly richer than standard assembly languages, and the type system does not *prevent* representing code with little type information. Second, it does not impose any particular runtime requirements or semantics on programs. Nevertheless, it's important to note that LLVM is *not intended to be a universal compiler IR*. In particular, LLVM does not represent high-level languages features directly (so it cannot be used for some language-dependent transformations), nor does it capture machine-dependent features or code sequences used by back-end code generators (it must be lowered to do so).

Because of the differing goals and representations, *LLVM is complementary to high-level virtual machines* (e.g., SmallTalk, Self, JVM, Microsoft's CLI, and others), and *not an alternative to these systems*. It differs from these in three key ways: (1) LLVM has no notion of high-level constructs such as classes, inheritance, or exception-handling semantics, even when compiling source languages with these features. (2) LLVM does not specify a runtime system or particular object model: it is low-level enough that the runtime system for a particular language can be implemented in LLVM itself. Indeed, LLVM can be used to *implement* high-level virtual machines. (3) LLVM does not guarantee type safety, memory safety, or language interoperability any more than the assembly language for a physical processor does.

The LLVM compilation framework exploits the code representation to provide a combination of five capabilities that we believe are important in order to support lifelong analysis and transformation for arbitrary programs. In general, these capabilities are quite difficult to obtain simultaneously, but the LLVM design does so inherently: (1) *Lifelong compilation model*: The compilation model allows sophisticated optimizations to be performed at all stages of an application's lifetime, including runtime and idle-time between runs, by preserving the LLVM code representation. (2) *Offline code generation*: Despite the previous point, it is possible to compile programs into efficient native machine code *offline*, using aggressive code generation techniques that are not suitable for runtime code generation. This is crucial for performance-critical programs, the most important target for lifelong optimization. (3) *User-based profiling and optimization*: The LLVM framework gathers profiling information at run-time *in the field* so that it is representative of actual users, and can apply it for profile-guided transformations both at run-time and in "idle-time"[1]. (4) *Transparent runtime model*: The system does not specify any particular object model, exception semantics, or runtime environment, thus allowing any language (or combination of languages) to be compiled to it. (5) *Uniform, whole-program compilation*: Language-independence makes it possible to optimize and compile all code comprising an application in a uniform manner (after linking), including language-specific runtime libraries and system libraries.

We believe that *no previously existing system provides all five of these properties*. Source-level compilers provide #2 and #4, but do not attempt to provide #1, #3 or #5. Link-time interprocedural optimizers, common in commercial compilers, provide the additional capability of #1 and #5 but only for link-time. Profile-guided optimizers for static languages provide benefit #2 at the cost of transparency, and most crucially do not provide #3. High-level virtual machines such as JVM or CLI provide #3 and most of #1, but do not provide #2, #4 or #5. Binary runtime optimization systems provide #2, #4 and #5, but provide #3 partially and do not provide #1. We explain these in more detail in Section 3.

We evaluate the effectiveness of the LLVM system with respect to three issues: (a) the size and effectiveness of the representation, including the ability to extract useful type information for C programs; (b) the compiler performance (not the performance of generated code which depends on the particular code generator or optimization sequences used); and (c) examples illustrating the key capabilities LLVM provides for several challenging compiler problems.

To summarize, the overall contributions of this work are as follows:

- LLVM defines a rich, low-level code representation with low-level operations and memory model, but with rich type, control-flow and dataflow information necessary for powerful, language-independent analyses and transformations. It achieves these goals through the novel instruction set features described earlier.

- The LLVM compiler framework provides transparent, lifelong analysis and optimization of arbitrary programs. The design provides all five capabilities we believe are important for effective lifelong code analysis and optimization of arbitrary programs, as listed above. To our knowledge, it is unique in this respect.

- Our experimental results show that the LLVM compiler can prove that an average of 74.6% of memory accesses are type-safe across a range of SPECINT 2000 C benchmarks. We also discuss based on our experience how the type information captured by LLVM is enough to safely perform a number of aggressive transformations that would traditionally be attempted only on type-safe languages in source-level compilers. We show that the LLVM representation is comparable in size to SPARC machine code (a RISC architecture) and roughly 25% larger than x86 code on average, despite capturing much richer type information as well as infinite register set in SSA form. Finally, we present example timings showing that the LLVM representation is amenable to extremely efficient interprocedural optimizations.

Our implementation of LLVM to date supports C and C++, which are traditionally compiled entirely statically. We are currently exploring whether LLVM can be beneficial for implementing dynamic runtimes such as JVM and CLI as well. LLVM is freely available under a non-restrictive license.[2]

The rest of this paper is organized as follows. Section 2 begins by describing the LLVM code representation. Section 3 then describes the design of the LLVM compiler framework. Section 4 discusses our evaluation of the LLVM system as described above. Section 5 compares LLVM with related previous systems. Section 6 concludes with a summary of the paper.

[1]The idle-time optimizer is planned, but has not yet been implemented.

[2]See the LLVM home-page: `http://llvm.cs.uiuc.edu/`.

2. PROGRAM REPRESENTATION

The LLVM representation is one of the key factors that differentiates LLVM from other systems. There are three specific design features in LLVM that we believe are novel: (1) The LLVM type system & `getelementptr` instruction (which implements type-safe pointer arithmetic). (2) The LLVM memory model, and (3) The `invoke` and `unwind` instructions, used to implement source-language exception handling features. This section of the paper gives an overview of the LLVM instruction set, describes these features, and briefly describes the offline and in-memory representations. The detailed syntax and semantics of the representation is defined in the LLVM reference manual [25].

2.1 Overview of the LLVM Instruction Set

The LLVM instruction set captures the key operations of ordinary processors but avoids machine-specific constraints such as physical registers, pipelines, and low-level calling conventions. LLVM provides an infinite set of typed virtual registers which can hold values of *primitive types* (boolean, integral, floating point, and pointers). The virtual registers are in Static Single Assignment (SSA) form [14]. LLVM is a load/store architecture: programs transfer values between registers and memory solely via `load` and `store` operations using typed pointers. The LLVM memory model is described in Section 2.3.

The entire LLVM instruction set consists of only 31 opcodes. This is possible because, first, we avoid multiple opcodes for the same operations[3]. Second, most opcodes in LLVM are overloaded (For example, the `add` instruction can operate on any integer or floating point operand type). Most instructions, including all arithmetic and logical operations, are in three-address form: they take one or two operands and produce a single result.

LLVM uses SSA form as its primary code representation, i.e., each SSA register is defined exactly once, and each use of a register is dominated by its definition. The LLVM instruction set includes an explicit `phi` instruction, which corresponds directly to the standard (non-gated) ϕ function of SSA form. SSA form simplifies many dataflow optimizations and enables fast, flow-insensitive algorithms to achieve many of the benefits of flow-sensitive algorithms without expensive dataflow analysis. It also dramatically simplifies many transformations because registers cannot have aliases.

LLVM also makes the Control Flow Graph (CFG) of every function explicit in the representation. A function is a set of basic blocks, and each basic block is a sequence of LLVM instructions, ending in exactly one terminator instruction (branches, return, `unwind`, or `invoke`; the latter two are explained later below). Each terminator explicitly specifies its successor basic blocks.

2.2 Language-independent Type Information, Cast and GetElementPtr

One of the fundamental design feature of LLVM is the inclusion of a language-independent type system. Every SSA register and explicit memory object has an associated type, and all operations obey strict type rules. This type information is used in conjunction with the instruction opcode to determine the exact semantics of an instruction (e.g. floating point vs. integer add). This type information enables a broad class of *high-level* transformations on *low-level* code (for example, see Section 4.1.1). In addition, type mismatches can be used to detect optimization bugs.

The LLVM type system includes source-language-independent primitive types with predefined sizes (void, bool, signed/unsigned integers from 8 to 64 bits, single- and double-precision floating-point types). This makes it possible (but not required) to write portable code using these types. LLVM also includes (only) four derived types: pointers, arrays, structures, and functions. We believe that most high-level language data types are eventually represented using some combination of these four types in terms of their operational behavior (e.g., C++ classes with inheritance are described in Section 4.1.2).

The LLVM '`cast`' instruction is used to convert a value of one type to another, arbitrary, type (needed for supporting non-type-safe languages, like C). The `cast` instruction is the *only* way to convert values: a program without `cast`s is necessarily type-safe (in the absence of memory access errors, e.g., array overflow [17]). LLVM types can also be used by aggressive interprocedural optimizations to check the correctness of a wide variety of transformations. Section 4.1.1 shows that despite allowing values to be arbitrarily cast to other types, reliable type information is actually available for most memory accesses in C programs compiled to LLVM.

A critical difficulty in preserving type information for low-level code is implementing address arithmetic. The `getelementptr` instruction is used by the LLVM system to perform type-safe pointer arithmetic. Given a typed pointer to an object of some aggregate type, this instruction calculates the address of a sub-element of the object in a type-preserving manner (effectively a combined '.' and '[]' operator for LLVM). For example, given a pointer to a structure and a field number, the `getelementptr` instruction returns a pointer to the field. Given a pointer to an array and an integer index, it returns a pointer to the specified element. Load and store instructions take a single pointer and do not perform any indexing. This design makes processing of memory accesses simple and uniform.

2.3 Explicit Memory Allocation and Unified Memory Model

LLVM provides instructions for typed memory allocation. The `malloc` instruction allocates one or more elements of a specific type on the heap, returning a typed pointer to the new memory. The `free` instruction releases memory allocated through `malloc`[4]. The `alloca` instruction is similar to `malloc` except that it allocates memory in the stack frame of the current function instead of the heap, and the memory is automatically deallocated on return from the function. All stack-resident data (including "automatic" variables) are allocated explicitly using `alloca`.

In LLVM, all addressable objects ("lvalues") are explicitly allocated. Global variable and function definitions define a symbol which provides the address of the object, not the object itself. This gives a unified memory model in which all memory operations, including call instructions, occur through typed pointers. There are no implicit accesses to

[3] For example, there are no unary operators: `not` and `neg` are implemented in terms of `xor` and `sub`, respectively.

[4] When native code is generated for a program, `malloc` and `free` instructions are converted to the appropriate native function calls, allowing custom memory allocators to be used.

memory, simplifying memory access analysis, and the representation needs no "address of" operator.

2.4 Function Calls and Exception Handling

For ordinary function calls, LLVM provides a `call` instruction that takes a typed function pointer (which may be a function name or an actual pointer value) and typed actual arguments. This abstracts away the calling conventions of the underlying machine and simplifies program analysis.

One of the most unusual features of LLVM is that it provides an explicit, low-level, machine-independent mechanism to implement exception handling in high-level languages. In fact, the same mechanism also supports `setjmp` and `longjmp` operations in C, allowing these operations to be analyzed and optimized in the same way that exception features in other languages are. The common exception mechanism is based on two instructions, `invoke` and `unwind`.

The `invoke` and `unwind` instructions together support an abstract exception handling model logically based on stack unwinding (though LLVM-to-native code generators may use either "zero cost" table-driven methods [8] or `setjmp`/`longjmp` to implement the instructions). The `invoke` instruction operates just like a `call` instruction, but associates an extra basic block with the call which is the starting block for an exception handler. When the program executes an `unwind` instruction, the program logically unwinds the stack until it removes an activation record created by an `invoke` instruction. It then transfers control to the basic block specified by the invoke. These two instructions explicitly expose exceptional control flow in the LLVM CFG, which is a critical aspect of its design.

These two primitive instructions can be used to implement a wide variety of exception handling mechanisms (we have currently implemented support for C's `setjmp`/`longjmp` calls and full support for the C++ exception model). The `unwind` instruction is used to implement exception throwing (and `longjmp`) in high-level languages. Code which must be executed if an exception is thrown (for example, `setjmp`, "catch" blocks, automatic variable destructors in C++, etc) uses the `invoke` instruction to stop unwinding, execute the desired code, then continue execution or unwinding as necessary.

```
{
  Class Object; // Has a destructor
  func();       // Might throw
  ...
}
```

Figure 1: C++ exception handling example

The example in Figure 1 illustrates a case where the `invoke` instruction is generated by a C++ front-end, in order to execute destructors of the stack-allocated `Object` if an exception is thrown as a result of the `func()` call. Figure 2 shows the LLVM code for the example. A front-end for Java would use similar code to unlock locks that are acquired through synchronized blocks or methods when exceptions are thrown. Finally, a `catch` clause (e.g., in C++, Java, OCaml, and other languages) would be implemented directly in terms of an exception destination. All of the exception semantics of the high-level language are written using runtime libraries that are called in the exception blocks.

2.5 Plain-text, Binary, and In-memory Representations

```
  ...
  ; Allocate stack space for object:
  %Object = alloca %Class, uint 1
  ; Construct object:
  call void %Class::Class(%Class* %Object)
  ; Call ``func()'':
  invoke void %func() to label %OkLabel
                  except label %ExceptionLabel
OkLabel:
  ; ... execution continues...
ExceptionLabel:
  ; If unwind occurs, excecution continues
  ; here.  First, destroy the object:
  call void %Class::~Class(%Class* %Object)
  ; Next, continue unwinding:
  unwind
```

Figure 2: LLVM code for the example

The LLVM representation is a *first class language* which defines equivalent textual, binary, and in-memory (i.e., compiler's internal) representations. The instruction set is designed to serve effectively both as a persistent, offline code representation and as a compiler internal representation, with no semantic conversions needed between the two[5]. Being able to convert LLVM code between these representations without information loss makes debugging transformations much simpler, allows test cases to be written easily, and decreases the amount of time required to understand the in-memory representation.

3. COMPILER ARCHITECTURE

The goal of the LLVM compiler framework is to enable sophisticated transformations at link-time, run-time, and after an application is installed in the field, by operating on the LLVM representation of a program at all stages. To be practical though, it must be transparent to application developers and end-users, and it must be efficient enough for use with real-world applications. This section describes how the overall system and the individual components are designed to achieve all these goals.

3.1 High-Level Design of the LLVM Compiler Framework

Figure 3 shows the high-level architecture of the LLVM system. Briefly, static compiler front-ends emit code in the LLVM representation, which is combined together by the LLVM linker. The linker performs a variety of link-time optimizations, then produces highly optimized native code for a given target (this step may alternatively be deferred to install time), and saves LLVM code with the native code. At runtime, a light-weight instrumentation system is used to detect program hot spots and perform simple runtime optimizations. The program behavior collected by the runtime optimizer can be collected and attached to the program, allowing an offline optimizer to perform aggressive profile-driven interprocedural optimizations *in the field* during idle-time, using profile information gathered from the end-user's and not the developer's runs.

This strategy provides five powerful benefits that are not available in the traditional model of static compilation to na-

[5]In contrast, typical JVM implementations convert from the stack-based bytecode language used offline to an appropriate representation for compiler transformations, and some even convert to SSA form for this purpose (e.g., [7]).

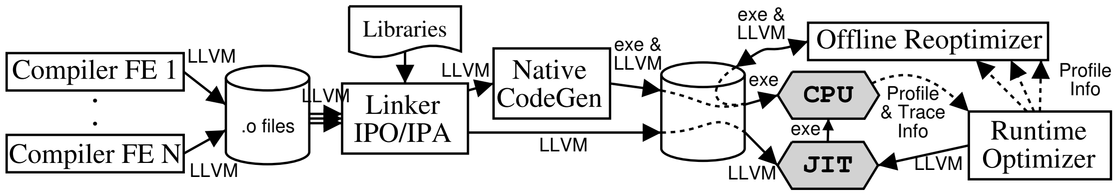

Figure 3: LLVM system architecture diagram

tive machine code. We argued in the Introduction that these capabilities are important for lifelong analysis and transformation, and we named them: (1) *lifelong compilation model*, (2) *offline code generation*, (3) *user-based profiling and optimization*, (4) *transparent runtime model*, and (5) *uniform, whole-program compilation*. These are difficult to obtain simultaneously for at least two reasons. First, offline code generation (#2) normally does not allow optimization at later stages on the higher-level representation instead of native machine code (#1 and #3). Second, lifelong compilation have traditionally been associated only with bytecode-based languages, which do not provide #4.

In fact, we noted in the Introduction that *no existing compilation approach provides all the capabilities listed above.* Our reasons are as follows:

- Traditional source-level compilers provide #2 and #4, but do not attempt #1, #3 or #5. They do provide interprocedural optimization, but require significant changes to application Makefiles.

- Several commercial compilers provide the additional benefit of #1 and #5 at link-time by exporting their intermediate representation to object files [19, 4, 23] and performing optimizations at link-time. No such system we know of is also capable of preserving its representation for runtime or offline use (benefits #1 and #3).

- Higher-level virtual machines like JVM and CLI provide benefit #3 and partially provide #1 (in particular, the need for bytecode verification greatly restricts the optimizations that may be done before runtime) and #5 (e.g., CLI can support code in multiple languages). They do not provide #2 because runtime optimization is generally only possible when using JIT code generation. They do not aim to provide #4, and instead provide a rich runtime framework for languages that match their runtime and object model, e.g., Java and C#.

- Transparent binary runtime optimization systems like Dynamo and the runtime optimizers in Transmeta processors provide benefits #2, #4 and #5, but not #1, and provide benefit #3 to a limited extent at runtime because they are constrained to work on native binary code, limiting the optimizations they can perform. Omniware [1] provides #2, #4 and perhaps #5, but does not provide a representation suitable for high-level analysis and optimization (i.e., #1).

- Profile Guided Optimization for static languages provide benefit #3 at the cost of not being transparent (they require a multi-phase compilation process). Additionally, PGO suffers from three main problems: (1) Empirically, developers are unlikely to use PGO, except when compiling benchmarks. (2) When PGO *is* used, the application is tuned to the behavior of the training run. If the training run is not representative of the end-user's usage patterns, the program may actually be pessimized by the profile information. (3) The profiling information is completely static, meaning that the compiler cannot make use of phase behavior in the program or adapt to changing usage patterns.

There are also limitations of the LLVM strategy. First, language-specific optimizations must be performed in the front-end before generating LLVM code. (LLVM is *not* designed to be a universal representation for source languages.) Second, it is an open question whether languages requiring sophisticated runtime systems such as Java can benefit directly from LLVM. We are currently exploring the potential benefits of implementing higher-level virtual machines such as JVM or CLI on top of LLVM.

The subsections below describe the key components of the LLVM compiler architecture, emphasizing design and implementation features that make the capabilities above practical and efficient.

3.2 Compile-Time: External front-end & static optimizer

External static LLVM compilers (referred to as front-ends) translate source-language programs into the LLVM virtual instruction set. Each static compiler can perform three key tasks, of which the first and third are optional: (1) Perform language-specific optimizations, e.g., optimizing closures in languages with higher-order functions. (2) Translate source programs to LLVM code, preserving as much useful type information for data values as possible. (3) Invoke LLVM passes for global or interprocedural optimizations at the module level. The LLVM optimizations are built into libraries, making it easy for front-ends to use them. The front-end does not have to perform SSA construction. Instead, variables can be allocated on the stack (which is not in SSA form), and the LLVM stack promotion pass can be used to build SSA form.

Note that many "high-level" optimizations are not language-dependent, and are often special cases of more general optimizations which may be performed on the LLVM level e.g., virtual function resolution for C++ as described in Section 4.1.2. If this is the case, it is often useful to extend the LLVM optimizer rather than investing effort in code which only benefits a particular front-end. This also allows the optimizations to be performed throughout the lifetime of the program.

3.3 Linker & Interprocedural Optimizer

Link time is the first phase of the compilation process where most[6] of the program is available for analysis and transformation. As such, link-time is a natural place to perform aggressive interprocedural optimizations across the entire program. The link-time optimizations operate on LLVM directly, and can take advantage of the semantic information it contains.

The design of compile- and link-time optimizers permit the use a well-known technique for speeding up interprocedural analysis: at compile-time, interprocedural summaries can be computed for each function in the program and attached to the LLVM bytecode. The link-time interprocedural optimizer can then process these interprocedural summaries as input instead of having to compute results from scratch. This technique can dramatically speed up incremental compilation when a small number of translation units are modified [6]. Note that this is achieved without building a program database or deferring the compilation of the input source code until link-time.

3.4 Offline or JIT Native Code Generation

After link-time optimization, a code generator is selected to translate from LLVM to native code for the current platform (we currently support the Sparc V9 and x86 architectures), in one of two ways. In the first configuration, the code generator is run offline (i.e., statically) at link time to generate high performance native code for the application, potentially expensive code generation techniques. If the user decides to use the post-link (runtime and offline) optimizers, a copy of the entire LLVM bytecode for the program is included into the executable itself[7]. In addition, the code generator inserts light-weight instrumentation into the program to identify frequently executed loop regions.

Alternatively, a Just-in-time Execution Engine can be used which invokes the appropriate code generator at runtime, translating a function at a time for execution (or uses a portable LLVM interpreter if no native code generator is available).

3.5 Runtime Path Profiling & Reoptimization

One of the goals of the LLVM project is to develop a new strategy for runtime optimization of ordinary applications. We briefly describe the strategy and then summarize its key benefits.

As a program executes, the most frequently executed execution paths are identified through a combination of offline and online instrumentation. The offline instrumentation is inserted just before native code generation (described above), identifying frequently executed loop regions in the code. An online instrumentation library then instruments such a hot loop region to identify frequently executed paths within that region. Once "hot" paths are identified, we duplicate the original LLVM code into a trace, perform LLVM optimizations on it, and then regenerate native code into a *software trace cache*. The native code is then stitched into the existing application code for subsequent executions.

The strategy described here is powerful because it combines the following three characteristics: (a) Native code generation can be performed ahead-of-time using sophisticated algorithms to generate high-performance code. (b) The native code generator and the runtime optimizer can work together since they are both part of the LLVM framework, allowing the runtime optimizer to exploit support from the code generator (e.g., for instrumentation and simplifying transformations). (c) The runtime optimizer can use high-level information from the LLVM representation to perform sophisticated runtime optimizations.

We believe these three characteristics together represent one "optimal" design point for a runtime optimizer because they allow the best choice in three key aspects: high-quality initial code generation (offline rather than online), cooperative support from the code-generator, and the ability to perform aggressive optimizations (by using LLVM rather than native code as the input).

3.6 Offline Reoptimization with End-user Profile Information

Some applications are not particularly amenable to runtime optimization: these applications often have a large amount of code, none of which is very "hot". Because of this, the runtime optimizer cannot afford to spend a large amount of time improving any one piece of the code, although it can still detect the most frequent paths executed by the program (for code layout optimizations).

In order to support these applications and to support other optimizations which require potentially expensive analyses or transformations, an offline reoptimizer can be used. It is designed to be run during idle time on the user's computer, allowing it to be much more aggressive than the runtime optimizer.

The offline reoptimizer combines profile information gathered by the runtime optimizer with the LLVM to optimize and recompile the application. In this way it is able to perform aggressive profile-driven interprocedural optimization without competing with the application for processor cycles. As the usage pattern of the application changes over time, the runtime and offline reoptimizers could coordinate to ensure the highest achievable performance.

4. APPLICATIONS AND EXPERIENCES

Sections 2 and 3 describe the design of the LLVM code representation and compiler architecture. In this section, we evaluate this design in terms of three categories of issues: (a) the effectiveness of the representation; (b) the speed of performing whole-program analyses and transformations in the compiler; and (c) illustrative uses of the LLVM system for challenging compiler problems, focusing on how the novel capabilities in LLVM benefit these uses.

4.1 Representation Issues

One of the key contributions of the LLVM representation is the language-independent type system. Does this type system provide any fundamental value when it can be violated with casts? Second, how do high-level language features (e.g., classes) map onto the LLVM type system and code representation? Third, how large is the LLVM representation when written to disk?

4.1.1 *What value does type information provide?*

Reliable type information about programs can enable the optimizer to perform aggressive transformations that would be difficult otherwise (such as reordering two fields of a

[6]Note that shared libraries and system libraries may not be available for analysis at link time, or may be compiled directly to native code.

[7]Eliminating the possibility that the runtime or offline optimizers will get the wrong bytecode for a given program.

structure or optimizing memory management [26]). However, since LLVM is a weakly-typed language, declared type information is not reliable and some analysis (typically including a pointer analysis) must check the declared type information before it can be used. A key question is how much *reliable* type information is available in programs compiled to LLVM?

LLVM includes a flow-insensitive, field-sensitive and context-sensitive points-to analysis called Data Structure Analysis (DSA) [27], and several transformations in LLVM use DSA as the main foundation (e.g., Automatic Pool Allocation [26]). As part of the analysis, DSA extracts LLVM types for memory objects in the program which it verified to be accessed in a type-safe manner. It does this by using the types present in the LLVM representation as speculative type information, and checks conservatively whether that type information is correct. DSA performs no type-inference: it only verifies that memory accesses agree with the declared types[8].

For a wide range of benchmarks, we measured the fraction of static load and store operations for which reliable type information about the accessed objects is available. Table 1 shows this statistic for the C benchmarks in SPEC CPU2000[9]. Simpler benchmarks (e.g., the Olden and Ptrdist benchmarks) had even better results, scoring close to 100% in most cases.

Benchmark Name	**Typed Accesses**	**Untyped Accesses**	**Typed Percent**
164.gzip	1674	15	99.1%
175.vpr	3986	400	90.9%
179.art	585	0	100.0%
181.mcf	581	0	100.0%
183.equake	881	48	94.8%
186.crafty	9849	603	94.2%
188.ammp	1570	3279	32.4%
197.parser	1532	2207	41.0%
254.gap	6578	15508	29.8%
255.vortex	15845	8725	64.5%
256.bzip2	1020	52	95.1%
300.twolf	7279	7249	50.1%
average			74.3%

Table 1: Loads and Stores which are provably typed

The table shows that many of these programs (164, 176, 179, 181, 183, 186, & 256) are surprisingly type-safe, despite the fact that the programming language does not enforce type-safety. The leading cause of loss of type-safety in the remaining programs is the use of custom memory allocators (i.e., 197, 254, 255, & 300) and DSA not being aggressive enough (in 188). Despite the use of custom allocators, DSA is still able to prove a significant number of accesses to be type-safe.

It is important to note that similar results would be very difficult to obtain if LLVM had been an untyped representation. As an example, an earlier version of the C-to-LLVM front-end was based on GCC's RTL internal representation, which provided little useful type information and both DSA and pool allocation were much less effective. Our new C front-end is based on the AST representation in the GCC front-end, which makes much more type information available.

4.1.2 How do high-level features map onto LLVM?

Compared to source languages, LLVM is a much lower level representation. Even C, which itself is quite low-level, has many features which must be lowered by a compiler targeting LLVM. For example, complex numbers, structure copies, unions, bit-fields, variable sized arrays, and `setjmp/longjmp` all must be lowered by an LLVM C compiler. In order for the representation to support effective analyses and transformations, the mapping from source-language features to LLVM should capture the high-level operational behavior as cleanly as possible.

We discuss this issue by using C++ as an example, since it is the richest language for which we have an implemented front-end. We believe that all the complex, high-level features of C++ are expressed clearly in LLVM, allowing their behavior to be effectively analyzed and optimized:

- Implicit calls (e.g. copy constructors) and parameters (e.g. 'this' pointers) are made explicit.

- Templates are fully instantiated by the C++ front end before LLVM code is generated.

- Base classes are expanded into nested structure types. For this C++ fragment:

  ```
  class base1 { int Y; };
  class base2 { float X; };
  class derived : base1, base2 { short Z; };
  ```

  the LLVM type for the "derived" class is '`{ {int}, {float}, short }`'. If the classes have virtual functions, a v-table pointer would also be included and initialized at object allocation time to point to the virtual function table, described below.

- A virtual function table is represented as a global, *constant* table of typed function pointers, plus the type-id object for the class. With this representation, virtual method call resolution can be performed by the LLVM optimizer as effectively as by a typical source compiler (more effectively if the source compiler uses only per-module instead of whole-program pointer analysis).

- C++ exceptions are lowered to the 'invoke' and 'unwind' instructions as described in Section 2.4, exposing exceptional control flow in the CFG. In fact, having this information available at link time enables LLVM to use an interprocedural analysis to eliminate unused exception handlers. This optimization is much less effective if done on a per-module basis in a source-level compiler.

We believe that similarly clean LLVM implementations exist for most constructs in other language families like Scheme, SmallTalk, the ML family, Java and Microsoft CLI (important examples include closures and continuations). We aim to explore this issue in the future and preliminary work is underway on the implementation of Java and Scheme front-ends.

[8]DSA is actually quite aggressive: it can often prove that objects stored into "generic" `void*` data structure (and then loaded from it later) are type-safe despite the casts to and from `void*`.

[9]Unfortunately, LLVM bugs prevented getting numbers for 176.gcc, 177.mesa, and 253.perlbmk in time for submission.

4.1.3 *How compact is the LLVM representation?*

Since code for the compiled program is stored in the LLVM representation throughout its lifetime, it is important that it not be overly large. The flat, three-address form of LLVM is well suited for a simple linear layout, with most instructions requiring only a single 32-bit word each in the file. Figure 4 shows the size of LLVM files for SPEC CPU2000 executables after linking, compared to native X86 and 32-bit Sparc executables produced by GCC 3.3 in both cases (using -O3).

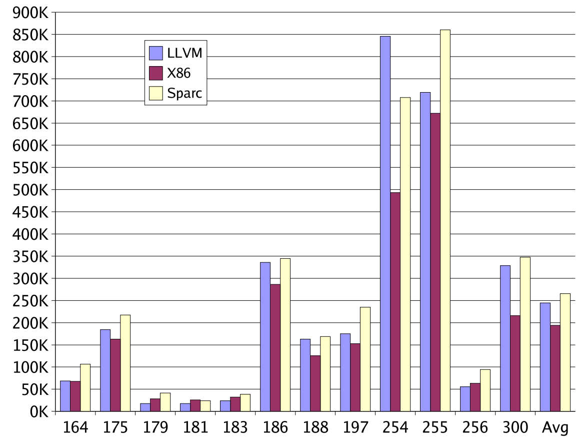

Figure 4: Executable sizes for LLVM, X86, Sparc

The figure shows that LLVM code is about the same size as native executables for SPARC, and is roughly 25% larger on average for x86 (a much denser, variable-size instruction set). We believe this is a very good result given that LLVM encodes rich type information, control flow information, and data-flow (SSA) information that native executables do not. We have not yet attempted to optimize the size of the bytecode files, so this may be further reduced in the future.

4.1.4 *How fast is LLVM?*

An important aspect of LLVM is that the low-level representation enables *efficient* analysis and transformation, because of the small, uniform instruction set, the explicit CFG and SSA representations, and careful implementation of data structures. This speed is important for uses "late" in the compilation process (i.e., at link-time or run-time). In order to provide a sense for the speed of LLVM, Figure 5 shows the table of runtimes for several interprocedural optimizations. All timings were collected on an 3.06GHz Intel Xeon processor.

The table includes numbers for **DGE** (aggressive[10] Dead global variable and function Elimination), **DAE** (an aggressive Dead Argument Elimination), **inline** (a function integration pass), **DSA** (Data Structure Analysis), and **GCC** (time to compile the programs with the gcc 3.3 compiler at -O3, provided as a reference point). All these interprocedural optimizations work on the whole program at link-time.

We find that in all cases, the optimization time is substantially less than that to compile the program with GCC, despite the fact that GCC does *no* cross module optimization, and very little interprocedural optimization within a translation unit. Optimizations such as DGE and DAE are very efficient, because they only have to touch small portions of the program to perform their analysis. The inlining pass is also fairly quick, taking time linear in the number of inlines it must perform (in 255.vortex, for example, it inlines 792 functions, deleting the bodies of 292 functions which are subsequently dead). Data Structure Analysis is the most aggressive analysis of the group, owing largely to the fact that it is a *context-sensitive*, field-sensitive, flow-insensitive pointer analysis, but it too is relatively fast compared with GCC.

[10] "Aggressive" DCE's all assume objects are dead until proven otherwise, allowing dead objects with cycles to be deleted.

Benchmark	DGE	DAE	inline	DSA	GCC
164.gzip	0.0052	0.0001	0.0241	0.0299	3.10
175.vpr	0.0001	0.0000	0.0045	0.0081	7.11
179.art	0.0005	0.0000	0.0096	0.0047	0.69
181.mcf	0.0005	0.0000	0.0202	0.0040	2.00
183.equake	0.0006	0.0000	0.0119	0.0054	0.91
186.crafty	0.0178	0.0005	0.0917	0.2709	10.98
188.ammp	0.0066	0.0003	0.1693	0.1115	6.81
197.parser	0.0088	0.0006	0.1888	0.2880	6.17
254.gap	0.0345	0.0034	0.3856	7.8341	20.08
255.vortex	0.0287	0.0073	1.2419	3.0451	24.15
256.bzip2	0.0022	0.0000	0.0322	0.0154	1.86
300.twolf	0.0128	0.0003	0.2580	0.1381	17.34

Figure 5: Interprocedural optimization timings (in seconds)

4.2 Applications using life-time analysis and optimization capabilities of LLVM

Finally, to illustrate the capabilities provided by the compiler framework, we briefly describe three examples of how LLVM has been used for widely varying compiler problems, using the novel capabilities described in the introduction.

4.2.1 *Projects using LLVM as a general compiler infrastructure*

As noted earlier, we have implemented several compiler techniques in LLVM. The most aggressive of these are Data Structure Analysis (DSA) and Automatic Pool Allocation [26], which analyze and transform programs in terms of their logical data structures. These techniques inherit a few significant benefits from LLVM, especially, (a) these techniques are only effective if most of the program is available, i.e., at link-time (b) type information allows is crucial for their effectiveness; (c) the techniques are source-language independent; and (c) SSA significantly improves the precision of DSA, which is flow-insensitive.

Other researchers not affiliated with our group have been actively using or exploring the use of the LLVM compiler framework, in a number of different ways. These include using LLVM as an intermediate representation for binary-to-binary transformations, as a compiler back-end to support a hardware-based trace cache and optimization system, as a basis for runtime optimization and adaptation of Grid programs, and as a compiler for memory partitioning and optimization of embedded codes.

4.2.2 *SAFECode: A safe low-level representation and execution environment*

SAFECode provides a "safe" code representation and execution environment, based on a type-safe subset of LLVM. The goal of our work is to enforce memory safety of programs in the SAFECode representation through static analysis, by using a variant of automatic pool allocation instead of garbage collection [17], and using extensive interprocedural static analysis to minimize runtime checks [24, 17].

The SAFECode system exploits nearly all capabilities of the LLVM framework, except runtime optimization. It directly uses the LLVM code representation, which provides the ability to analyze C and C++ programs, which is crucial for supporting embedded software, middle-ware, and system libraries. SAFECode relies on the type information in LLVM (with no syntactic changes) to check and enforce type safety. It relies on the array type information in LLVM to enforce array bounds safety, and uses interprocedural analysis to eliminate runtime bounds checks in many cases [24]. It uses interprocedural safety checking techniques, exploiting the link-time framework to retain the benefits of separate compilation (a key difficulty that led many related systems to avoid using interprocedural techniques [16, 21]).

4.2.3 External ISA design for Virtual Instruction Set Computers

Virtual Instruction Set Computers are processor designs that use distinct instruction set architectures for the external program representation (the virtual ISA or V-ISA) and for the actual hardware ISA in a particular implementation (I-ISA). A software translator co-designed with the hardware (essentially, a sophisticated, implementation-specific back-end compiler) translates V-ISA code to the I-ISA, and is the only software that is aware of the I-ISA.

In recent work, we argued that an extended version of the LLVM instruction set could be a good choice for the external V-ISA for such processor designs [2]. We proposed a novel implementation strategy for the virtual-to-native translator that enables offline code translation and caching of translated code in a completely OS-independent manner[11].

That work *exploits* the important features of the instruction set representation, and extends it to be suitable as a V-ISA for hardware. The fundamental benefit of LLVM for this work is that the LLVM code representation is low-level enough to represent arbitrary external software (including operating system code), yet provides rich enough information to support sophisticated compiler techniques in the translator. A second key benefit is the ability to do both offline and online translation, which is exploited by the OS-independent translation strategy.

5. RELATED WORK

We focus on comparing LLVM with three classes of previous work: other virtual machine-based compiler systems, research on typed assembly languages, and link-time or dynamic optimization systems.

As noted in the Introduction, the goals of LLVM are complementary to those of higher-level language virtual machines such as SmallTalk, Self, JVM, and Microsoft CLI. We noted three key differences in the goals of LLVM vs. these systems, and we do not repeat them here. The Omniware virtual machine [1] is perhaps the most similar to LLVM: they use an abstract low-level RISC architecture and can support multiple source languages. However, the goals of their work are to provide safety, not performance. In particular, they do not include high-level information necessary for lifelong optimization.

There has also been a a wide range of work in the field of typed intermediate representations. Functional languages often use strongly typed intermediate languages (e.g. [31]) as a natural extension of the source language. Projects on typed assembly languages (e.g., TAL [28] and LTAL [9]) focus on preserving high-level type information and type safety during compilation and optimizations. The SafeTSA [3] representation is a combination of type information with SSA form, which aims to provide a safe but more efficient representation than JVM bytecode for Java programs. In contrast, the LLVM virtual instruction set does not attempt to preserve type safety of high-level languages, to capture high-level type information from such languages, or to enforce code safety directly (though it can be used to do so). Instead, the goal of LLVM is to enable sophisticated analyses and transformations beyond static compile time.

There have been attempts to define a unified, generic, intermediate representation. These have largely failed, ranging from the original UNiversal Computer Oriented Language [33] (UNCOL), which was discussed but never implemented, to the more recent Architecture and language Neutral Distribution Format [13] (ANDF), which was implemented but has seen limited use. These unified representations attempt to describe programs at the AST level, implying that they must include features from all possible source languages. LLVM is much less ambitious and is more like an assembly language: it uses a small set of types and low-level operations, and the "implementation" of high-level language features is described in terms of these types. In some ways, LLVM simply appears as a very strict RISC architecture.

Kistler and Franz describe a compilation architecture for performing optimization in the field, using simple initial load-time code generation, followed by profile-guided runtime and offline optimization. Their system uses Slim Binaries [20] as its code representation, a very compact tree-based code representation. Kistler and Franz observe that other representations could be used. LLVM could be directly used within this architecture, making it unnecessary to regenerate SSA form for every recompilation, as they suggest.

Several systems perform interprocedural optimization at link-time. Some operate on assembly code for a given processor [29, 32, 12, 30] (focusing primarily on machine-dependent optimizations), while others export additional information from the static compiler, either in the form of an IR or annotations) [34, 19, 4, 23]. None of these approaches attempt to support optimization at runtime or offline after software is installed in the field, and would also be difficult to directly extend to do so.

There have also been several systems that perform transparent runtime optimization of native code [5, 18, 15]. These systems inherit all the challenges of optimizing machine-level code [29] in addition to the constraint of operating under the tight time constraints of runtime optimization. In contrast, LLVM aims to provide type, dataflow (SSA) information, and an explicit CFG for use by runtime optimizations. For example, our online tracing framework (Section 3.5) directly exploits the CFG to perform limited instrumentation of hot loop regions at runtime. Finally, none of these systems supports link-time, install-time, or offline optimizations, with or without profile information.

[11]The goals and contributions are quite different from this work, though both are based on the LLVM instruction set.

6. CONCLUSION

This paper has described LLVM, a system for performing lifelong code analysis and optimization. The system that we describe uses a low-level (but fully typed) language as the representation of a program throughout its lifetime. Because the LLVM representation is language independent, all of the code for a program, including system libraries and portions written in multiple languages, can be compiled and optimized together. The LLVM compiler framework includes a powerful link-time interprocedural optimizer, a low-overhead tracing technique for runtime trace-based optimization, and both Just-In-Time and static code generators.

We showed experimentally and based on experience that LLVM makes available extensive type information even for C programs, which can be used to safely perform a number of aggressive transformations that would normally be attempted only on type-safe languages in source-level compilers. We also showed that the LLVM representation is comparable in size to SPARC machine code and only 25% larger than x86 code on average, despite capturing much richer type information as well as an infinite register set in SSA form. Finally, we showed that several example whole-program optimizations can be performed very fast on the LLVM representation. A key question we are exploring currently is whether high-level language virtual machines can be implemented effectively on top of the LLVM runtime optimization and code generation framework.

7. REFERENCES

[1] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 127–136. ACM Press, 1996.

[2] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. A Low-level Virtual Instruction Set Architecture. page (to appear), San Diego, CA, Dec 2003.

[3] W. Amme, N. Dalton, M. Franz, and J. ery. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *PLDI*, June 2001.

[4] A. Ayers, S. de Jong, J. Peyton, and R. Schooler. Scalable cross-module optimization. *ACM SIGPLAN Notices*, 33(5):301–312, 1998.

[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI*, pages 1–12, June 2000.

[6] M. Burke and L. Torczon. Interprocedural optimization: eliminating unnecessary recompilation. *TOPLAS*, 15(3):367–399, 1993.

[7] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *Java Grande*, pages 129–141, 1999.

[8] D. Chase. Implementation of exception handling. *The Journal of C Language Translation*, 5(4):229–240, June 1994.

[9] J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound TAL for back-end optimization. In *PLDI*, San Diego, CA, Jun 2003.

[10] A. Chernoff, et al. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, 1998.

[11] R. Cohn, D. Goodwin, and P. Lowney. Optimizing Alpha executables on Windows NT with Spike. *Digital Technical Journal*, 9(4), 1997.

[12] R. Cohn, D. Goodwin, P. Lowney, and N. Rubin. Spike: An optimizer for Alpha/NT executables, 1997.

[13] A. Consortium. The Architectural Neutral Distribution Format, `http://www.andf.org/`.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, pages 13(4):451–490, October 1991.

[15] J. C. Dehnert, et al. The Transmeta Code Morphing Software: Using speculation, recovery and adaptive retranslation to address real-life challenges. In *Proc. 1st IEEE/ACM Symp. Code Generation and Optimization*, San Francisco, CA, Mar 2003.

[16] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *PLDI*, Snowbird, UT, June 2001.

[17] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *LCTES*, San Diego, CA, Jun 2003.

[18] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA*, pages 26–37, 1997.

[19] M. F. Fernández. Simple and effective link-time optimization of Modula-3 programs. *ACM SIGPLAN Notices*, 30(6):103–115, 1995.

[20] M. Franz and T. Kistler. Slim binaries. *Communications of the ACM*, 40(12), 1997.

[21] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *PLDI*, Berlin, Germany, June 2002.

[22] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 168–181. ACM Press, 2003.

[23] IBM Corp. XL FORTRAN: Eight Ways to Boost Performance. White Paper, 2000.

[24] S. Kowshik, D. Dhurjati, and V. Adve. Ensuring code safety without runtime checks for real-time control systems. In *CASES*, Grenoble, France, Oct 2002.

[25] C. Lattner and V. Adve. LLVM Language Reference Manual. `http://llvm.cs.uiuc.edu/docs/LangRef.html`.

[26] C. Lattner and V. Adve. Automatic Pool Allocation for Disjoint Data Structures. In *Proc. ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, Jun 2002.

[27] C. Lattner and V. Adve. Data Structure Analysis: A Fast and Scalable Context-Sensitive Heap Analysis. Tech. Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Apr 2003.

[28] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *TOPLAS*,

21(3):528–569, May 1999.

[29] R. Muth. *Alto: A Platform for Object Code Modification*. Ph.d. Thesis, Department of Computer Science, University of Arizona, 1999.

[30] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proc. USENIX Windows NT Workshop*, August 1997.

[31] Z. Shao, C. League, and S. Monnier. Implementing Typed Intermediate Languages. In *International Conference on Functional Programming*, pages 313–323, 1998.

[32] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, Dec. 1992.

[33] T. Steel. Uncol: The myth and the fact. *Annual Review in Automated Programming 2*, 1961.

[34] D. Wall. Global register allocation at link-time. In *Proc. SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, 1986.